Python
Lecture 4

Loops

Chapter 1

What are Loops?

In our everyday lives, we tend to repeat a lot of processes without noticing.

For instance, if we want to cook a delicious recipe, we might have to prepare our ingredients by chopping them up. We chop and chop and chop until all of our ingredients are the right size. At this point, we stop chopping.

If we break down our chopping task into a series of three smaller steps, we have:

1. An initialization: We're ready to cook and have a collection of ingredients we want to chop. We will start at the first ingredient.
2. A repetition: We're chopping away. We are performing the action of chopping over and over on each of our ingredients, one ingredient at a time.
3. An end condition: We see that we have run out of ingredients to chop and so we stop.

In programming, this process of using an initialization, repetitions, and an ending condition is called a *loop*. In a loop, we perform a process of *iteration* (repeating tasks).

Programming languages like Python implement two types of iteration:

1. *Indefinite iteration*, where the number of times the loop is executed depends on how many times a condition is met.
2. *Definite iteration*, where the number of times the loop will be executed is defined in advance (usually based on the collection size).

Typically we will find loops being used to iterate a collection of items. In the above example, we can think of our ingredients we want to chop as our collection. This is a form of definite iteration since we know how long our collection is in advance and thus know how many times we need to iterate over the collection of ingredients.

Some collections might be small — like a short string, while other collections might be massive like a range of numbers from 1 to 10,000,000! But don't worry, loops give us the ability to masterfully handle both ends of the spectrum. This simple, but powerful, concept saves us a lot of time and makes it easier for us to work with large amounts of data.

In this lesson, we'll learn how to use Python to implement both definite and indefinite iteration in our own programs.

Why Loops?
Before we get to writing our own loops, let's explore what programming would be like if we couldn't use loops.

Let's say we have a list of `ingredients` and we want to print every element in the list:

```
ingredients = ["milk", "sugar", "vanilla extract", "dough",
"chocolate"]
```

If we only use `print()`, our program might look like this:

```
print(ingredients[0])
print(ingredients[1])
print(ingredients[2])
print(ingredients[3])
print(ingredients[4])
```

The output would be:

```
milk
sugar
vanilla extract
dough
chocolate
```

That's still manageable, We're writing 5 `print()` statements (or copying and pasting a few times). Now imagine if we come back to this program and our list had 10, or 24601, or ... 100,000,000 elements? It would take an extremely long time and by the end, we could still end up with inconsistencies and mistakes.

Don't dwell too long on this tedious scenario — we'll learn how loops can help us out in the next exercise. For now, let's gain an appreciation for loops.

Chapter 2

For Loops: Introduction
Now that we can appreciate what loops do for us, let's start with your first type of loop, a `for` loop, a type of definite iteration.

In a `for` loop, we will know in advance how many times the loop will need to iterate because we will be working on a collection with a predefined length. In our examples, we will be using Python lists as our collection of elements.

With `for` loops, on each iteration, we will be able to perform an action on each element of the collection.

Before we work with any collection, let's examine the general structure of a `for` loop:

```
for <temporary variable> in <collection>:
  <action>
```

Let's break down each of these components:

1. A `for` keyword indicates the start of a `for` loop.
2. A `<temporary variable>` that is used to represent the value of the element in the collection the loop is currently on.
3. An `in` keyword separates the temporary variable from the collection used for iteration.
4. A `<collection>` to loop over. In our examples, we will be using a list.
5. An `<action>` to do anything on each iteration of the loop.

Let's link these concepts back to our `ingredients` example. This `for` loop prints each `ingredient` in `ingredients`:

```
ingredients = ["milk", "sugar", "vanilla extract", "dough",
"chocolate"]

for ingredient in ingredients:
  print(ingredient)
```

In this example:

1. `ingredient` is the `<temporary variable>`.
2. `ingredients` is our `<collection>`.
3. `print(ingredient)` was the `<action>` performed on every iteration using the temporary variable of `ingredient`.

This code outputs:

```
milk
sugar
vanilla extract
dough
chocolate
```

Some things to note about `for` loops:

- **Temporary Variables:**

  A temporary variable's name is arbitrary and does not need to be defined beforehand. Both of the following code snippets do the exact same thing as our above example:

  ```
  for i in ingredients:
    print(i)
  for item in ingredients:
   print(item)
  ```

  Programming best practices suggest we make our temporary variables as descriptive as possible. Since each iteration (step) of our loop is accessing an ingredient it makes more sense to call our temporary variable `ingredient` rather than `i` or `item`.
- **Indentation:**

  Notice that in all of these examples the `print` statement is indented. Everything at the same level of indentation after the `for` loop declaration is included in the loop body and is run on every iteration of the loop.

  ```
  for ingredient in ingredients:
    # Any code at this level of indentation
    # will run on each iteration of the loop
    print(ingredient)
  ```

  If we ever forget to indent, we'll get an `IndentationError` or unexpected behavior.
- **Elegant loops:**

  Python loves to help us write elegant code so it allows us to write simple `for` loops in one-line. In order to see the below example as one line, you may need to expand your narrative window. Here is the previous example in a single line:

  ```
  for ingredient in ingredients: print(ingredient)
  ```

  Note: One-line `for` loops are useful for simple programs. It is not recommended you write one-line loops for any loop that has to perform

multiple complex actions on each iteration. Doing so will hurt the readability of your code and may ultimately lead to buggier code.

Let's practice writing our own `for` loop!

Example

```
board_games = ["Settlers of Catan", "Carcassone", "Power Grid", "Agricola", "S
crabble"]

sport_games = ["football", "hockey", "baseball", "cricket"]

for game in board_games:
  print(game)

for sport_game in sport_games:
  print(sport_game)
```

For Loops: Using Range

Often we won't be iterating through a specific list (or any collection), but rather only want to perform a certain action multiple times.

For example, if we wanted to print out a `"Learning Loops!"` message six times using a `for` loop, we would follow this structure:

```
for <temporary variable> in <list of length 6>:
  print("Learning Loops!")
```

Notice that we need to iterate through a list with a length of six, but we don't necessarily care what is inside of the list.

To create arbitrary collections of any length, we can pair our `for` loops with the trusty Python built-in function `range()`.

An example of how the `range()` function works, this code generates a collection of 6 integer elements from `0` to `5`:

```
six_steps = range(6)

# six_steps is now a collection with 6 elements:
# 0, 1, 2, 3, 4, 5
```

We can then use the range directly in our `for` loops as the collection to perform a six-step iteration:

```
for temp in range(6):
  print("Learning Loops!")
```
Would output:

```
Learning Loops!
Learning Loops!
Learning Loops!
Learning Loops!
Learning Loops!
Learning Loops!
```
Something to note is we are not using `temp` anywhere inside of the loop body. If we are curious about which loop iteration (step) we are on, we can use `temp` to track it. Since our range starts at `0`, we will add `+ 1` to our `temp` to represent how many iterations (steps) our loop takes more accurately.

```
for temp in range(6):
  print("Loop is on iteration number " + str(temp + 1))
```
Would output:

```
Loop is on iteration number 1
Loop is on iteration number 2
Loop is on iteration number 3
Loop is on iteration number 4
Loop is on iteration number 5
Loop is on iteration number 6
```
Let's try out using a range in a `for` loop!

Example

```
promise = "I will finish the python loops module!"


for x in range(5):
  print(promise)
```

Chapter 3

While Loops: Introduction

In Python, `for` loops are not the only type of loops we can use. Another type of loop is called a `while` loop and is a form of indefinite iteration.

A `while` loop performs a set of instructions as long as a given condition is true.

The structure follows this pattern:

```
while <conditional statement>:
  <action>
```

Let's examine this example, where we print the integers `0` through `3`:

```
count = 0
while count <= 3:
  # Loop Body
  print(count)
  count += 1
```

Let's break the loop down:

1. `count` is initially defined with the value of `0`. The conditional statement in the `while` loop is `count <= 3`, which is true at the initial iteration of the loop, so the loop body executes.

   Inside the loop body, `count` is printed and then incremented by `1`.
2. When the first iteration of the loop has finished, Python returns to the top of the loop and checks the conditional again. After the first iteration, `count` would be equal to `1` so the conditional still evaluates to `True` and so the loop continues.
3. This continues until the `count` variable becomes `4`. At that point, when the conditional is tested it will no longer be `True` and the loop will stop.

The output would be:

```
0
1
2
3
```

Note the following about `while` loops before we write our own:

- **Indentation:**

  Notice that in our example the code under the loop declaration is indented. Similar to a `for` loop, everything at the same level of indentation after the `while` loop declaration is run on every iteration of the loop while the condition is true.

```
while count <= 3:
  # Loop Body
  print(count)
  count += 1
  # Any other code at this level of indentation will
  # run on each iteration
```

If we ever forget to indent, we'll get an `IndentationError` or unexpected behavior.

- **Elegant loops:**

  Similar to `for` loops, Python allows us to write elegant one-line `while` loops. Here is our previous example in a single line:

  ```
  count = 0
  while count <= 3: print(count); count += 1
  ```

  Note: Here we separate each statement with a `;` to denote a separate line of code.

Let's practice writing a `while` loop!

Example

```
countdown = 10
while countdown >= 0:
  print(countdown)
  countdown -= 1
print("We have liftoff!")
```

While Loops: Lists

A `while` loop isn't only good for counting! Similar to how we saw `for` loops working with lists, we can use `while` loops to iterate through a list as well.

Let's return to our ingredient list:

```
ingredients = ["milk", "sugar", "vanilla extract", "dough",
"chocolate"]
```

We know that `while` loops require some form of a variable to track the condition of the loop to start and stop.

Take some time to think about what we would use to track whether we need to start/stop the loop if we want to iterate through `ingredients` and print every element.

Click here to find out!

We know that a list has a predetermined length. If we use the length of the list as the basis for how long our `while` loop needs to run, we can iterate the exact length of the list.

We can use the built-in Python `len()` function to accomplish this:

```
# Length would be equal to 5
length = len(ingredients)
```

We can then use this `length` in addition to another variable to construct the `while` loop:

```
length = len(ingredients)
index = 0

while index < length:
  print(ingredients[index])
  index += 1
```

Let's break this down:

```
# Length will be 5 in this case
length = len(ingredients)
```

Explanation
As mentioned, we need a way to know how many times we need our loop to iterate based on the size of the collection.

This comes in the form of our `length` variable which stores the value of the length of the list.

```
# Index starts at zero
index = 0
```

Explanation
We still need an additional variable that will be used to compare against our `length`.

```
while index < length:
```

Explanation
In our `while` loop conditional, we will compare the `index` variable to the length of our list stored inside of the `length` variable.

On the first iteration, we will be comparing the equivalent of `0 < 5` which will evaluate to `True`, and start the execution of our loop body.

```
# The first iteration will print ingredients[0]
print(ingredients[index])
```

Explanation
Inside of our loop body, we can use the `index` variable to access our `ingredients` list and print the value at the current iteration.

Since our `index` starts at zero, our first iteration will print the value of the element at the zeroth index of our `ingredients` list, then the next iteration will print the value of the element at the first index, and so on.

```
# Increment index to access the next element in ingredients
# Each iteration gets closer to making the conditional no longer true
index += 1
```

Explanation
On each iteration of our `while` loop, we need to also increment the value of `index` to make sure our loop can stop once the `index` value is no longer smaller than the `length` value.

This increment also helps us access the next value of the `ingredients` list on the next iteration.

Our final output would be:

```
milk
sugar
vanilla extract
dough
chocolate
```

Let's use a `while` loop to iterate through some lists!

Example

```
python_topics = ["variables", "control flow", "loops", "modules", "classes"]

#Your code below:
length = len(python_topics)
index = 0

while index < length:
  print(f"I am learning about {python_topics[index]}")
  index += 1
```

Chapter 4

Infinite Loops
We've iterated through lists that have a discrete beginning and end. However, let's consider this example:

```
my_favorite_numbers = [4, 8, 15, 16, 42]

for number in my_favorite_numbers:
  my_favorite_numbers.append(1)
```

Take some time to ponder what happens with this code.

Click to see what would happen!
Every time we enter the loop, we add a `1` to the end of the list that we are iterating through. As a result, we never make it to the end of the list. It keeps growing forever!

A loop that never terminates is called an *infinite loop*. These are very dangerous for our code because they will make our program run forever and thus consume all of your computer's resources.

A program that hits an infinite loop often becomes completely unusable. The best course of action is to avoid writing an infinite loop.

Note: If you accidentally stumble into an infinite loop while developing on your own machine, you can end the loop by using control + c to terminate the program. If you're writing code in our online editor, you'll need to **refresh the page** to get out of an infinite loop.

Loop Control: Break
Loops in Python are very versatile. Python provides a set of control statements that we can use to get even more control out of our loops.

Let's take a look at a common scenario that we may encounter to see a use case for *loop control statements*.

Take the following list `items_on_sale` as our example:

```
items_on_sale = ["blue shirt", "striped socks", "knit dress", "red headband", "dinosaur onesie"]
```

It's often the case that we want to search a list to check if a specific value exists. What does our loop look like if we want to search for the value of `"knit dress"` and print out `"Found it"` if it did exist?

It would look something like this:

```
for item in items_on_sale:
  if item == "knit dress":
    print("Found it")
```
This code goes through each `item` in `items_on_sale` and checks for a match. This is all fine and dandy but what's the downside?

Once `"knit_dress"` is found in the list `items_on_sale`, we don't need to go through the rest of the `items_on_sale` list. Unfortunately, our loop will keep running until we reach the end of the list.

Since it's only 5 elements long, iterating through the entire list is not a big deal in this case but what if `items_on_sale` had 1000 items? What if it had 100,000 items? This would be a huge waste of time for our program!

Thankfully you can stop iteration from inside the loop by using `break` loop control statement.

When the program hits a `break` statement it immediately terminates a loop. For example:

```
items_on_sale = ["blue shirt", "striped socks", "knit dress", "red
headband", "dinosaur onesie"]

print("Checking the sale list!")

for item in items_on_sale:
  print(item)
  if item == "knit dress":
    break

print("End of search!")
```
This would produce the output:

```
Checking the sale list!
blue shirt
striped socks
knit dress
End of search!
```
When the loop entered the `if` statement and saw the `break` it immediately ended the loop. We didn't need to check the elements of `"red headband"` or `"dinosaur onesie"` at all.

Now let's `break` some loops!

Example

```
dog_breeds_available_for_adoption = ["french_bulldog", "dalmatian", "shihtzu",
 "poodle", "collie"]
dog_breed_I_want = "dalmatian"

for dog_breed in dog_breeds_available_for_adoption:
  print(dog_breed)
  if dog_breed == dog_breed_I_want:
    break

print("They have the dog I want!")
```

Loop Control: Continue

While the `break` control statement will come in handy, there are other situations where we don't want to end the loop entirely. What if we only want to skip the current iteration of the loop?

Let's take this list of integers as our example:

```
big_number_list = [1, 2, -1, 4, -5, 5, 2, -9]
```

What if we want to print out all of the numbers in a list, but only if they are positive integers. We can use another common loop control statement called `continue`.

```
for i in big_number_list:
  if i <= 0:
    continue
  print(i)
```

This would produce the output:

```
1
2
4
5
2
```

Notice a few things:

1. Similar to when we were using the `break` control statement, our `continue` control statement is usually paired with some form of a conditional (`if`/`elif`/`else`).
2. When our loop first encountered an element (`-1`) that met the conditions of the `if` statement, it checked the code inside the block and saw the `continue`. When the loop then encounters a `continue` statement it immediately skips the current iteration and moves onto the next element in the list (`4`).

3. The output of the list only printed positive integers in the list because every time our loop entered the `if` statement and saw the `continue` statement it simply moved to the next iteration of the list and thus never reached the `print` statement.

Let's `continue` learning about control statements with some exercises!

Example

```
ages = [12, 38, 34, 26, 21, 19, 67, 41, 17]

for age in ages:
  if age < 21:
    continue
  print(age)
```

Nested Loops

Loops can be nested in Python, as they can with other programming languages. We will find certain situations that require nested loops.

Suppose we are in charge of a science class, that is split into three project teams:

```
project_teams = [["Ava", "Samantha", "James"], ["Lucille", "Zed"], ["Edgar", "Gabriel"]]
```

Using a for or while loop can be useful here to get each team:

```
for team in project_teams:
  print(team)
```

Would output:

```
["Ava", "Samantha", "James"]
["Lucille", "Zed"]
["Edgar", "Gabriel"]
```

But what if we wanted to print each individual student? In this case we would actually need to *nest* our loops to be able loop through each sub-list. Here is what it would look like:

```
# Loop through each sublist
for team in project_teams:
  # Loop elements in each sublist
  for student in team:
    print(student)
```

This would output:

```
Ava
Samantha
James
Lucille
Zed
Edgar
Gabriel
```

Let's practice writing a nested loop!

Example

```
sales_data = [[12, 17, 22], [2, 10, 3], [5, 12, 13]]

scoops_sold = 0

for location in sales_data:
  print(location)
  for element in location:
    scoops_sold += element

print(scoops_sold)
```

Chapter 5

List Comprehensions: Introduction

So far we have seen many of the ideas about using loops in our code. Python prides itself on allowing programmers to write clean and elegant code. We have already seen this with Python giving us the ability to write `while` and `for` loops in a single line.

In this exercise, we are going to examine another way we can write elegant loops in our programs using *list comprehensions*.

To start, let's say we had a list of integers and wanted to create a list where each element is doubled. We could accomplish this using a `for` loop and a new list called `doubled`:

```
numbers = [2, -1, 79, 33, -45]
doubled = []

for number in numbers:
  doubled.append(number * 2)

print(doubled)
```

Would output:

```
[4, -2, 158, 66, -90]
```

Let's see how we can use the power of list comprehensions to solve these types of problems in one line. Here is our same problem but now written as a list comprehension:

```
numbers = [2, -1, 79, 33, -45]
doubled = [num * 2 for num in numbers]
print(doubled)
```

Let's break down our example in a more general way:

```
new_list = [<expression> for <element> in <collection>]
```

In our `doubled` example, our list comprehension:

1. Takes an element in the list `numbers`
2. Assigns that element to a variable called `num` (our `<element>`)
3. Applies the `<expression>` on the element stored in `num` and adds the result to a new list called `doubled`
4. Repeats steps 1-3 for every other element in the `numbers` list (our `<collection>`)

Our result would be the same:

```
[4, -2, 158, 66, -90]
```

Example

```
grades = [90, 88, 62, 76, 74, 89, 48, 57]
scaled_grades = [num + 10 for num in grades]
print(scaled_grades)
```

List Comprehensions: Conditionals

List Comprehensions are very flexible. We even can expand our examples to incorporate conditional logic.

Suppose we wanted to double only our negative numbers from our previous `numbers` list.

We will start by using a `for` loop and a list `only_negative_doubled`:

```
numbers = [2, -1, 79, 33, -45]
only_negative_doubled = []

for num in numbers:
  if num < 0:
    only_negative_doubled.append(num * 2)

print(only_negative_doubled)
```

Would output:

```
[-2, -90]
```

Now, here is what our code would look like using a list comprehension:

```
numbers = [2, -1, 79, 33, -45]
negative_doubled = [num * 2 for num in numbers if num < 0]
print(negative_doubled)
```

Would output the same result:

```
[-2, -90]
```

In our `negative_doubled` example, our list comprehension:

1. Takes an element in the list `numbers`.
2. Assigns that element to a variable called `num`.
3. Checks if the condition `num < 0` is met by the element stored in `num`. If so, it goes to step 4, otherwise it skips it and goes to the next element in the list.
4. Applies the expression `num * 2` on the element stored in `num` and adds the result to a new list called `negative_doubled`
5. Repeats steps 1-3 (and sometimes 4) for every other element in the `numbers` list.

We can also use If-Else conditions directly in our comprehensions. For example, let's say we wanted to double every negative number but triple all positive numbers. Here is what our code might look like:

```
numbers = [2, -1, 79, 33, -45]
doubled = [num * 2 if num < 0 else num * 3 for num in numbers ]
print(doubled)
```

Would output:

```
[6, -2, 237, 99, -90]
```

This is a bit different than our previous comprehension since the conditional `if num < 0 else num * 3` comes after the expression `num * 2` but before our `for` keyword.

Let's write our own list comprehensions with conditionals!

Example

```
heights = [161, 164, 156, 144, 158, 170, 163, 163, 157]
can_ride_coaster = [height for height in heights if height > 161]
print(can_ride_coaster)
```